08 — Memória e Estado

🕒 estado da arte 2026-07 · revisão 2026-07-26 · 📖 ~13 min de leitura

Objetivos de aprendizagem

Ao final deste capítulo, você deve ser capaz de:

1. **Distinguir** as três camadas do problema — estado de sessão, memória de longo prazo e estado do workspace — e o requisito próprio de cada uma;
2. **Explicar** por que memória **não é** RAG (Retrieval-Augmented Generation) (memória = retrieval + caminho de escrita + gestão de estado) e por que markdown versionável venceu bancos vetoriais no domínio de código;
3. **Derivar** uma política de recall a partir da fórmula recência × importância × relevância, e uma de esquecimento a partir do uso;
4. **Avaliar** o impacto da **reversibilidade** (checkpoint de workspace) sobre o cálculo de risco de permissões;
5. **Implementar** a persistência de sessão do harness-zero (adapter SQLite + `/resume`) na etapa 4.

O problema

O modelo esquece tudo entre chamadas; o harness lembra por ele. "Memória e estado" cobre três camadas com requisitos diferentes:

1. **Estado de sessão** — a conversa em si: mensagens, tool-calls, metadados. Precisa sobreviver a reinícios e permitir retomar (`resume`), ramificar e reverter.
2. **Memória de longo prazo** — fatos que atravessam sessões: preferências do usuário, decisões do projeto, aprendizados. Precisa ser **selecionável** (nem tudo entra em todo contexto) e **atualizável** (fatos mudam).
3. **Estado do workspace** — o que o agente *fez* nos arquivos. Precisa ser **reversível**: desfazer as mudanças de um agente é tão importante quanto fazê-las.

A tese que unifica as três: a janela de contexto é memória volátil e cara; tudo o que precisa durar vive **fora** dela, e o harness decide o que trazer de volta e quando.

Fundamentos científicos

A memória de agentes tem literatura madura — e ela dá o vocabulário exato para o que os harnesses fazem na prática.

- **A janela como RAM** — MemGPT: LLMs as Operating Systems, arXiv 2310.08560 trata o contexto como memória principal escassa, apoiada por dois níveis externos (*recall* de histórico recente e *archival* pesquisável), com o **agente** paginando dados via tool calls ("context page faults"). Decisão: quem decide o que despejar e o que buscar é o agente, não um pipeline RAG fixo.

- **A taxonomia canônica** — CoALA, arXiv 2309.02427 separa memória **episódica** (experiência passada), **semântica** (conhecimento do mundo/usuário) e **procedural** (habilidades/código), mais a working memory. Decisão: no momento da escrita, decida *que tipo* de memória aquele fato é — cada tipo se recupera diferente. O survey de mecanismos de memória, arXiv 2404.13501 (depois ACM TOIS) organiza o subsistema por *fontes* · *formas* · *operações* (escrita, gestão/consolidação, leitura) — orce esforço por operação, não só pelo índice de busca.
- **A fórmula de recall** — Generative Agents, arXiv 2304.03442 (UIST '23) guarda observações num *memory stream* datado e recupera por um score composto de **recência × importância × relevância** (decaimento exponencial de recência, importância pontuada por LLM (Large Language Model), relevância por embedding). É a fórmula concreta que um harness deve implementar para rankear o que reentra no contexto — e introduz a **consolidação por reflexão** (sintetizar reflexões de alto nível a partir de clusters de observações).
- **Esquecimento controlado** — MemoryBank, arXiv 2305.10250 (AAAI '24) decai/reforça a força de cada memória por uma curva de Ebbinghaus (tempo decorrido × frequência de acesso), mantendo o store limitado. Decisão: memória não-usada é candidata a poda — o *tracking de uso* é o que fecha o ciclo.
- **Memória como aprendizado** — Reflexion, arXiv 2303.11366 (NeurIPS '23) converte feedback de resultado em auto-reflexão verbal, persistida num buffer episódico e reinjetada na próxima tentativa — melhorar sem atualizar pesos. E arquiteturas recentes (A-MEM, arXiv 2502.12110; Mem0, arXiv 2504.19413) tratam a escrita como um pipeline *extrair → consolidar → linkar*, com a rede de memórias se auto-organizando (estilo Zettelkasten). Ponte para o cap. 16 (auto-melhoria).

(Bibliografia completa e ponteiros: `livro/bibliografia.md`.)

Fontes da indústria

- **Sessão como log de eventos durável** — Manage sessions (Claude Code): cada sessão é gravada continuamente em disco como **JSONL** por projeto (uma linha por mensagem/tool-use/metadado); `--continue` retoma a mais recente no diretório, `--resume` abre um seletor. Decisão: "retomar" é **restaurar estado completo** (tool calls, resultados, modo de permissão, objetivo ativo), não replay de texto — o harness é dono de um log durável privado, não de um schema público estável.
- **Reversão do workspace como trilha separada** — Checkpointing (Claude Code) captura o estado do código antes de cada prompt; `/rewind` restaura código, conversa **ou** ambos (100 checkpoints recentes, limpos com a sessão). O file-checkpointing do Agent SDK expõe isso como primitiva reusável. Decisão: desfazer o *código* é um store separado de desfazer a *conversa*, ligados pelo índice do prompt.
- **Memória durável como arquivos com precedência** — How Claude remembers your project: a hierarquia CLAUDE.md (política gerenciada → usuário → projeto → local), o atalho `#` para anexar uma linha de memória, `/memory` para editar. Decisão: memória cross-sessão é **markdown em tiers de precedência** (o mais específico vence) — versionável, auditável, escopada; relida no launch como contexto sempre-ligado.
- **A memory tool (beta) e "assuma interrupção"** — Memory tool: o modelo pede operações (`view/create/str_replace/…`) num diretório `/memories` que persiste entre conversas, mas a execução é **client-side** — seu app implementa o armazenamento (e a proteção contra path traversal,

limites de tamanho, expiração). O sistema injeta "ASSUMA INTERRUPÇÃO: sua janela pode ser resetada a qualquer momento". Pareada com o context management (context editing evicta pares stale da janela; a memory tool persiste fora dela) — dois níveis: higiene de curto prazo + store externo de longo prazo. Decisão: para agentes de longa duração, você precisa dos dois; a janela é efêmera, o `/memories` é a fonte de verdade (o padrão do ensaio harnesses para agentes de longa duração: log de progresso estruturado, lido no início e atualizado no fim de cada sessão).

- **Memória ≠ RAG** — a distinção virou tese de indústria: a Letta ("RAG is not agent memory") e a AWS Bedrock AgentCore argumentam que RAG é leitura *stateless*; memória é leitura **+ caminho de escrita + gestão de estado** (admissão, resolução de fatos conflitantes, invalidação). A Letta expõe *memory blocks* auto-editáveis e tiers **core/recall/archival** (a hierarquia do MemGPT como produto); a mem0 roteia cada fato por camada com tempo de vida próprio; a Zep/Graphiti modela memória como **grafo de conhecimento bi-temporal** (fatos desatualizados são *invalidados*, não deletados); a LangMem/LangGraph separa **short-term (thread)** de **long-term (store por namespace)**. Decisão: não dá para "comprar" memória pregando um vector store — é preciso um pipeline de escrita/atualização/invalidação.
- **Consulte também**: a coleção viva Awesome Harness Engineering — Memory & State reúne mais recursos consultáveis desta dimensão (padrões, artigos e implementações), curados por problema.

O estado da arte

1. Três camadas, três campeões — e nenhum banco vetorial

As três camadas do problema receberam campeões diferentes na coorte: **estado de sessão** (durabilidade de banco — opencode com SQLite + eventos replayáveis; Codex com rollout jsonl por turno; OpenHands com event-stream), **memória de longo prazo** (relevância + rigor de formato — OpenHarness com memdir versionado, `relevance.py` e `usage.py`; Hermes com `MEMORY.md`/`USER.md` + `session_search`), **estado de workspace** (reversão via git — Aider e gemini-cli). E o achado que persiste: **nenhum dos harnesses de código usa banco vetorial** para memória. No domínio de código, markdown versionável venceu embeddings — porque memória de código precisa de *caminho de escrita* (o agente edita o arquivo) e auditabilidade, exatamente o que a tese "memória ≠ RAG" prevê.

2. A fórmula de recall e o esquecimento saíram do paper para o código

O `relevance.py` + `usage.py` do OpenHarness é a instância prática do stream de Generative Agents: seleciona por relevância o que entra no contexto e marca uso — memória não-usada vira candidata a poda (a curva de esquecimento do MemoryBank, na prática). O Hermes formaliza a **manutenção ativa**: um único tool edita `MEMORY.md`/`USER.md` com **nudges periódicos** (a cada 10 turnos) e um `session_search` (índice FTS5/BM25 sobre o SQLite de sessões, com modos discovery/recall/sumarização) dá **recall cross-session** — a camada archival do MemGPT construída sobre busca textual, não vetorial.

3. Reversibilidade virou primitiva — e muda o cálculo de risco

O checkpoint de workspace deixou de ser feature e virou primitiva: o **Aider** foi pioneiro anos atrás (estado git-nativo: auto-commit atômico por rodada, `aider_commit_hashes`, `/undo`, `.aider.chat.history.md`), o **gemini-cli** consagrou (`/restore`, `/rewind` do disco via snapshots git), e o Claude Code o expõe como checkpointing com trilhas separadas para código e conversa. A consequência de projeto é a mais interessante: **um agente cujas ações são reversíveis muda o cálculo de risco de tudo o mais** — permissões podem ser mais frouxas quando desfazer é barato (liga ao cap. 07).

4. Providers plugáveis e o harness como servidor de memória

A fronteira emergente: memória como serviço plugável. O Hermes já aceita provedores externos (Honcho, mem0, supermemory) por trás da sua camada; produtos como Letta/mem0/Zep se posicionam como a "camada de memória universal" consumível por qualquer harness. A tensão de projeto para as próximas rodadas: manter a memória como **arquivo local versionável** (auditável, portável, sem dependência) ou terceirizá-la para um store gerenciado (grafo bi-temporal, escala). No código, o arquivo ainda vence; fora dele, o pêndulo é menos claro.

> **LEITURA EXECUTIVA**
>
> O que está mais moderno: a moldura de tiers OS (RAM ↔ recall ↔ archival) com o agente paginando; recall por recência×importância×relevância com esquecimento por uso; reversão do workspace como primitiva que afrouxa permissões; e a distinção dura memória × RAG (write path + invalidação). **O que roubar:** persista a sessão como log de eventos durável (retomada = restaurar estado, não replay); trate memória como markdown versionável com tracking de uso; separe a trilha de reversão do código da conversa; e, para agentes longos, escreva um log de progresso durável assumindo que a janela some a qualquer momento.

Mão na massa — harness-zero, etapa 4

A etapa 4 (`harness-zero/etapas/04-sessoes/`) dá persistência ao harness-zero: um **adapter SQLite** por trás de uma `StorePort` guarda mensagens e tool-calls como linhas tipadas, e `/resume` restaura o estado completo de uma sessão anterior (não só o texto). Fiel ao hexagonal *por refatoração*: a dor que faz a porta nascer é reabrir o processo e perder a conversa. Exercício de completude: a persistência cobre o *happy path*; você adiciona um `USER.md`/`MEMORY.md` mínimo lido no início e um log de progresso atualizado ao fim — o padrão "assuma interrupção" na sua forma mais simples.

Verificação

1. Por que memória de agente não é a mesma coisa que RAG, e o que isso explica sobre a escolha de markdown versionável em vez de banco vetorial nos harnesses de código? (Memória = retrieval + caminho de escrita + gestão/invalidação de estado; código precisa de write path auditável.)

2. Você tem 10.000 memórias e espaço para 20 no contexto. Que score usa para escolher, e como decide o que podar com o tempo? (Recência × importância × relevância; poda por falta de uso — curva de esquecimento.)
3. Seu agente ganhou checkpoint de workspace com `/rewind`. Que decisão *de outra dimensão* isso permite afrouxar, e por quê? (Permissões — o cálculo de risco cai quando desfazer é barato; cap. 07.)

Apêndice A — Como cada repositório trata memória e estado

> Evidência por harness, com paths — complementação online, expandida a cada rodada.

opencode (rodada 1) — estado como banco de dados

Persistência em **SQLite via Drizzle** (`packages/core/database`, `core/session/sql.ts`): sessões, mensagens e partes são linhas tipadas. Sessões têm `parentID` (hierarquia para subagentes), suportam revert (`session/revert.ts`) e **compartilhamento** (`share/`, `sync/`). A V2 (`CONTEXT.md`) leva o desenho a "infra de dados": inbox durável de prompts, eventos replayáveis com cursores (`sessions.events({sessionID, after})`), snapshots de contexto persistidos entre reinícios. O modelo de estado mais robusto da rodada 1 — o harness como sistema distribuído com estado durável.

gemini-cli (rodada 1) — o workspace reversível

Memória de longo prazo nos próprios `GEMINI.md` (tool `save_memory`, global em `~/.gemini` + índice de projeto, com auto-memory testada em evals). O recurso distintivo é o **checkpointing baseado em git** (`services/gitService.ts` + `chatRecordingService.ts`): snapshots do workspace antes de edições, habilitando `/restore` e `/rewind` — desfazer as mudanças do agente no disco, não só na conversa — além de `/resume`.

OpenHarness (rodada 1) — memória como arquivo, com disciplina

`src/openharness/memory/` (13 módulos): memória persistente em markdown (`MEMORY.md`/memdir por projeto) com **schema versionado, escrita atômica com file-lock e assinaturas**. `relevance.py` seleciona o que entra no contexto; `usage.py` marca uso (memória não usada é candidata a poda). Sessões persistidas com metadados ricos (`services/session_storage.py`): modo de permissão, estado de arquivos lidos, skills invocadas, checkpoints de compactação. Retomada via `-c/--continue`, `-r/--resume`, `/resume`.

Aider (rodada 2) ⭐ estado git-nativo — o pioneiro da reversão

`aider/repo.py`: **auto-commit atômico por rodada** com mensagem gerada por LLM, atribuição de autoria configurável, `aider_commit_hashes` rastreando o que a IA fez, `dirty_commit` isolando mudanças pendentes. `/undo`, `diff` e `blame` viram a interface de memória; complementos

`.aider.chat.history.md` e `--restore-chat-history`. **Antecipou em anos** o "checkpoint git" que o gemini-cli e o Claude Code consagraram.

Hermes (rodada 2) ⭐ memória multicamada com recall cross-session

`MEMORY.md` (notas do agente) + `USER.md` (perfil do usuário) editados por tool única com **nudges periódicos** (a cada 10 turnos); provedores externos plugáveis (**Honcho, mem0, supermemory**); e **`session_search`** — índice FTS5 sobre o SQLite de sessões com três modos (discovery/BM25, recall janelado, sumarização por LLM) para recall cross-session. A camada archival do MemGPT sobre busca textual.

Codex CLI (rodada 2) — rollout jsonl por turno

Cada turno é persistido em **rollout jsonl** (recuperável); `SessionTask` (Regular/Review/Compact/UserShell) organiza a máquina de tarefas. Estado de sessão durável e resumível integrado ao loop (`core/src/session/`).

OpenHands (rodada 2) — event-stream persistido

`openhands/app_server/event/` persiste cada `Event` como JSON por conversa, com paginação, filtros e export de trajetória. O control-plane consome/persiste eventos; o loop ação-observação roda no SDK. Event-sourcing como coluna vertebral do estado.

OpenClaw (rodada 2) — session lanes e arquivos de workspace

Runs serializados por *session lane* com write-lock file-based entre processos; arquivos de workspace (`MEMORY.md`, `USER.md`, `IDENTITY.md`…) injetados com orçamentos (20k chars/arquivo, 60k total) e truncamento marcado. Persistência de conversa por canal.

ohmo (rodada 2) — backends de sessão/memória como plugins

Implementa `SessionBackend` e `MemoryCommandBackend` do OpenHarness como plugins de primeira classe (sem tocar no core), mais um **pool multi-sessão** (`RuntimeBundle` por `session_key`, recriado quando o cwd muda). Prova de que a fronteira app/engine foi desenhada.

IronClaw (rodada 2) — estado resumível por checkpoints

Estado resumível por **checkpoints**; máquina de estados Queued → Running → Blocked → Completed com **leases/heartbeats** e "one active run per canonical thread". O `LoopExit` carrega apenas referências duráveis — o loop nunca muta estado; o `LoopExitApplier` valida evidência host-owned antes de aplicar.

n8n (rodada 2) — memória do motor de workflow

Memória via *memory sub-nodes* (janela `contextWindowLength`, corte `maxTokensFromMemory`); estado do workflow persistido pelo motor entre execuções. Curto por natureza — execuções acionadas por evento não acumulam contexto longo (compactação nota 1, por design).

Frameworks (rodada frameworks)

LangGraph: **checkpointer** (short-term, thread-scoped) + **store** por namespace (long-term cross-thread); LangMem: memórias semântica/episódica/procedural como tools; Agents SDK e CrewAI: estado de sessão/curto-prazo com hooks de persistência. A distinção short × long term é primitiva de framework — o que os harnesses de código implementam à mão, os frameworks expõem como API.